Context-Aware Service Selection Based on the Augmented Reality Toolkit

Systementwicklungsprojekt

Technische Universität München
Chair for Applied Software Engineering

Felix Löw
loew@in.tum.de

January 13, 2004

This paper deals with extending the AR Toolkit's functionality to allow not only small stationary setups but wide range tracking applications as well. Based on a visionary scenario of a mobile user walking around in a building with an intelligent environment, we split the AR Toolkit's functionality in several components based on the DWARF (Distributed Wearable Augmented Reality Framework) system. Using a thin mobile client we describe a mechanism for a dynamic configuration according to the location context during runtime. This means that the system triggers a context change with different strategies and then gets reconfigured by the environment. We discuss the architecture of the location component, evaluate several strategies for a context change and show how environmental conditions can influence the quality of the system.

Contents

1 Context Awareness and Augmented Reality

This thesis deals with extending the ARToolkit for a new application domain. We want to use the ARToolkit marker detection ability to build a wide range tracking system. A special focus is how it can be managed to reconfigure a mobile setup according to a certain context situation during runtime. First we want to give a small overview over the terms context awareness and augmented reality. Then we give a short introduction into the DWARF framework(Distributed Wearable Augmented Reality Framework). As the system was built on the DWARF middleware the basic mechanisms are explained. This work was part of a group effort within the ARCHIE project. The basic features of ARCHIE are described at the end of this chapter.

1.1 Introduction to Context and Augmented Reality

Compared to virtual environments, which simulate a whole virtual world, Augmented Reality (AR) applications try to include virtual objects in the real environment. In order to augment a virtual object into a current environment a system has to be aware of the current context. What is context?

Here is a definition by Anind Dey [2]:

> Context is any information that can be used to characterize the situation of an entity. An entity is a person, place or object that is considered relevant to the interaction between a user and an application, including the user and application themselves.

Said in simple words context is every information about the environment that is needed by the system to perform its tasks. This kind of information could be considered as implicit input (not visible to the user) for the system. An Augmented Reality system expects two kinds of input:

- Explicit input by the user
  This could be simple keyboard input, mouse movements or speech recognition. The user has to know how the input should be entered and the input must be provided in an explicit way.
- Implicit input
  The system gathers information about the environment. The system behaves differently depending on the current situation a user is in. This kind of input is much more interesting, the system has to track the context and then it has to convert the raw data of the current context into high level information and perform action depending on it. The user does not have to know much about the system.

  Here are some examples what context could be:

- location
- time
- temperature
- people and objects nearby
- the user (preferences, history, knowledge, age, emotional state ...)
- available bandwidth
- sources in general

This work concentrates on indoor location awareness. The main idea is the premise that we have a large building with different infrastructures in every room. The user walks through this building and is able to use the infrastructure of the current room. The user wears a mobile computing system that is able to connect with the current network environment. But therefore his system has to be aware about the room and its infrastructure. Consider a mp3 player installed in a room. The user enters the room and gets a message that a player is now available for use. The user starts a song with his input device (e.g a PDA, or a mobile phone using Bluetooth). If the user exits the room the location context changes and he looses the ability of playing mp3 songs. But he might be able to do something else now.

So we have to use a technology that allows us to track the location in an indoor environment. For outdoor tracking GPS is a standard technology. But it does not work inside buildings. Tracking the location of a person inside a building is much more difficult. This work is an approach to realize location awareness inside a building by developing an optical marker detection service. This detection component is based on the Augmented Reality Toolkit (ARToolkit [5]) library.

First this thesis gives a short introduction into the DWARF system [1], then it will continue with an overview of the ARToolkit. After that it will go on with the different services realizing the tracking. The thesis will finish with a summary and some ideas for future improvements.

1.2 The DWARF-System

One requirement for this thesis was to realize the location aware module in the DWARF framework. Here are some basic mechanisms that are used by the services implemented in this project.

The DWARF-Framework [3] is a CORBA-based framework developed by the chair of Applied Software Engineering at the Technische Universität München (TU München). It is an approach to build augmented reality applications using independent services that can be combined to AR applications. The services can run on different machines in the network. The main idea is that these services connect dynamically according to their configuration.

A DWARF-service has:

- Needs
  Needs are special properties of DWARF services in order to request a certain functionality from all the other services running in the system. To satisfy this request the service

needs data or information from other communication partners. This needed information is described in the need property

- Abilities
  According to the need properties there are ability properties. This specification describes a functionality or information a service can provide to other services.

If a service has a certain ability to deliver some kind of data and another service has a need for exactly that kind of information they get connected by the servicemanager module. Then the two services can communicate directly via different communication protocols:

Method Calls: A service exports a certain method and this method could be called by a connected service importing the method (CORBA RPC).

Events: A service sends events to another service (CORBA notification service).

Shared Memory: A certain shared memory block could be accessed by services. One service provides data and writes it into the shared memory block, another service reads the data.

A servicemanager is running on every network node and handles the discovery and connection of individual services.

Here is a small example of two services communicating with each other (see figure 1.1):

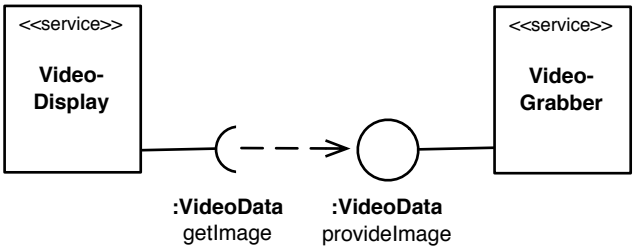

Figure 1.1: Example: Providing a video image

The service VideoGrabber has one ability of type "VideoData". This ability has the name "provideImage". The service "VideoDisplay" has the corresponding need "VideoData". The servicemananger connects both services and now the two services are able to communicate directly via the shared memory protocol. These abilities and needs can change during the runtime. If the "VideoDisplay" service loses its need for "VideoData" the connection breaks.

The configuration of each service is done by a service description in XML (see figure 1.2). It is possible to assign attributes to abilities and predicates to needs. This is shown in the service descriptions of the two services:

As you can see in the service description of the VideoDisplay the need for "VideoData" has the predicate "(type=head)". This means that the service will only be connected to a service providing "VideoData" if the corresponding ability carries the attribute (name="type" value="head"). This could be the case if the VideoDisplay service only displays images provided by a head camera. It will not be connected to a service that provides images from a

```
<service name="VideoGrabber">
    <ability name="provideImage" type="VideoData">
        <attribute name="type" value="head">
        <connector protocol="Shmem"/>
    </ability>
</service>

<service name="VideoDisplay">
    <need name="getImage" type="VideoData" predicate="(type=head)">
        <connector protocol="Shmem"/>
    </need>
</service>
```

Figure 1.2: DWARF service-description

camera attached to a breast. These predicates and attributes can also change dynamically during the runtime of an application.

Another mechanism uses the so called * - attributes [6]. It is used for a dynamic configuration of services during the runtime and will be described later with the concrete example of the dynamic configuration of the location awareness component.

1.3 Requirements for the ARCHIE-scenario

This work was done as one part of a group effort. The idea of the ARCHIE (Architectural Collaborative Home Improvement Environment) project [4] was to develop an application for architects which use augmented reality while building a model of the planned building. The ARCHIE system consists of several features:

- ARCHIE supports the process of designing new buildings
  Architects are able to design new buildings using an augmented reality system. With tangible devices like a little wall, they can put the real small wall and place it on the working desk, submit and now they receive a virtual wall connected to the location of the input device. With this method they can stuck walls together and build a new house.

- ARCHIE involves not only architects, but all stakeholders
  For the use of presentations to all stakeholders the system provides a presentation feature. All decision makers or customers are able to examine the virtual model of the building on a white board.

- ARCHIE allows collaborative design with multiple participants
  The design process can be done by several architects cooperating with each other at the same working desk or even at different locations.

- ARCHIE supports data logging for usability evaluations
  In order to improve user interfaces such as different kind of menus and as a result improve the performance of the user, the actions of current users are logged and evaluated.

- ARCHIE supports the integration of context aware application selection within a single runtime environment
  According to the current environment it is possible to change the application within runtime. For example if an architect wants to present the virtual building to the customers he enters the presentation room and the system automatically starts the application for the presentation. So the user is mobile and carries a wearable device that dynamically configures to the current environment.

This thesis deals with the mobile and context aware part of ARCHIE. A user can switch to other applications while entering new context situations without having the whole application logic on his wearable device. An example would be if an architect finishes his work on the model and exists the room and enters now the presentation room: the design session would be stopped and the presentation would be started. All the applications are running in the environment and their functionality are provided to the user. This is one attempt to let the vision of Ubiquitous Computing (Marc Weiser [10]) become more real.

2 The AR-Toolkit

The location aware module is based on optical marker detection. We have used the ARToolkit detection functionality for our purpose. In this chapter we try to layer the ARToolkit and describe the basic features of it.

2.1 Introduction to the AR-Toolkit

The Augmented Reality Toolkit is a well known software library written in C for optical marker detection. It has been developed by Hirokazu Kato at the University of Washington in Seattle. The Toolkit consists of different components depending on each other. So it is easy to reuse these components but also to build new components funding on the library functions. It is possible to write own applications with the Toolkit.

In this thesis it is split in components according to [8]:

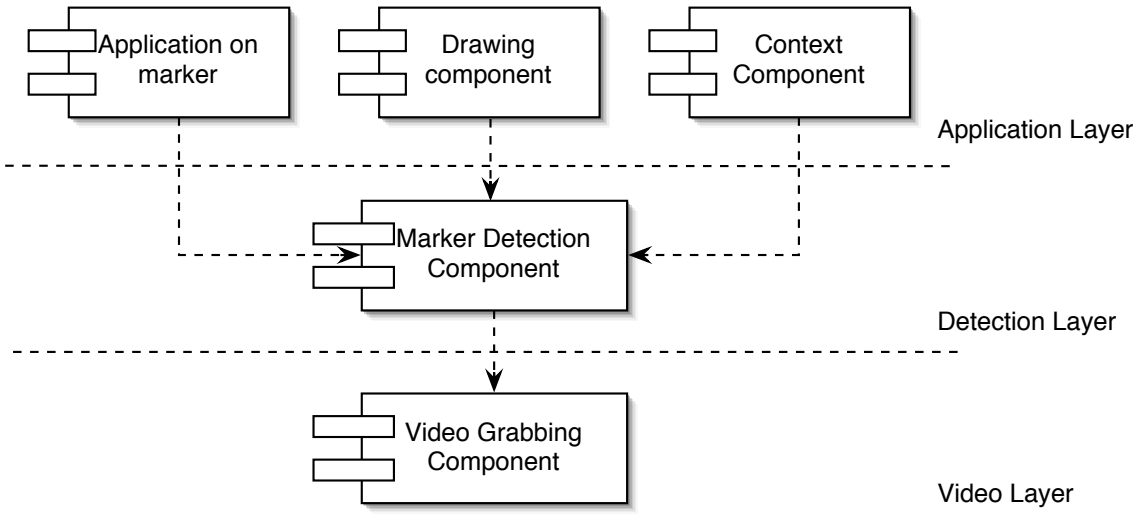

Figure 2.1: Layering of the ARToolkit

2.1.1 Video Layer

This layer provides video images to the detection layer. The realization of the "Videograbbing" depends on the platform and the hardware infrastructure. As we have worked on a Linux-system using a IEEE1394 firewire connection the software is based on the library

libdc1394. The library provides functions to establish a video connection and to grab video images. These functions are shown later.

The ARToolkit also provides other libraries for other platforms (e.g. Apple: Quicktime) and the platform could be selected before compilation. We used a simple iBot-camera for video capturing. The video layer we have implemented is completely independent from the ARToolkit library.

2.1.2 Detection Layer

This layer receives a video image from the bottom layer. With image detection methods the video image is checked if a marker becomes visible. These markers are black and white colored rectangles and triangles ordered in a certain way (see figure 2.2).

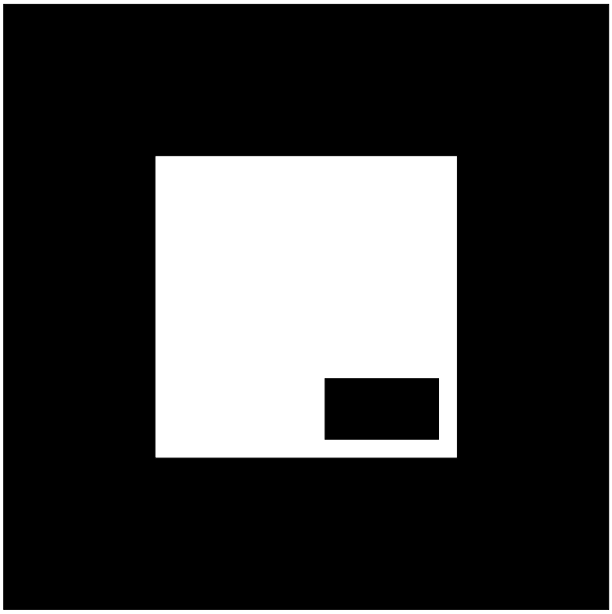

Figure 2.2: ARToolkit Marker

Of course the MarkerDetection component has to be configured with the markers being in use first. The ARToolkit library provides a tool to create your own markers as well.

2.1.3 Application Layer

It is possible to realize different types of application in the application layer. The ARToolkit itself does not provide all of the possibilities described here. It offers functions to draw VRML-scenes on the marker but also OpenGL drawing is supported. Therefore it calculates the camera position in order to display the model according to the right viewpoint (see figure 2.3).

We have put a new component on top of the other two layers. First we will introduce it as a context component. Task of this component is to transfer events from the detection layer

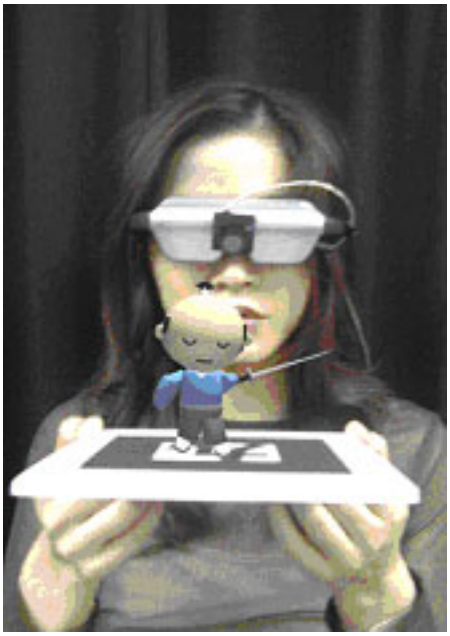

Figure 2.3: ARToolkit Application

into high level context information.

We try to develop a wide area tracking system using the toolkit. Originally this is not the idea of the ARToolkit. In this work we will not use it to display 3D models on a marker. We need only the detection capability of the toolkit to receive raw tracking data. The hardware infrastructure that is needed for this system is cheaper and easier to integrate than other indoor tracking systems. We only need a mobile setup and a video camera. The challenge with this idea of a wide area tracker is to have a dynamic configuration for the system. According to the current situation the system gets configured.

2.2 ARToolkit Environment

In this work we use only the basic functions of the ARToolkit. These functions are:

- Initializing the camera
- Load and unload patterns
- Detect markers

Other features are explained in the ARToolkit manual [5] and are described in the ar.h header file.

3 The Location Module

In this chapter we will describe the location module. First we will explain the system as a whole and how the different components interact with each other. Then we will go into the different components and explain their internal behavior and structure. Therefore we make a difference between components that are necessary for a dynamic configuration and those that are necessary for tracking.

3.1 Location-Awareness in DWARF

As we have explained the basic DWARF features and the functionality of the ARToolkit we now try to develop a location awareness module to estimate the room a user is in. First we have to split the ARToolkit into DWARF services and then we have to consider how we would realize the transformation of raw marker events into room information. Another question arises as we proceed: How is it possible to have the right configuration of the system according to the current room? And how is it possible to trigger a context change.

We already talked about the layering of the ARToolkit. The first attempt is to match each layer to a single DWARF-service:

ARToolkit Layer	**Function**	**DWARF-Service**
VideoGrabbing Component	provide video images via a IEEE1934 firewire connection	VideoGrabber
MarkerDetection Component	search for markers in the video image	MarkerDetection
Context Component	transform raw marker-events into high level room information	Context Estimator

Table 3.1: Matching ARToolkit layers on DWARF-Services

We have 3 DWARF services depending on each other now:

- VideoGrabber
  This service is independent from the ARToolkit. It uses the libdc1394 library in order to grab video images from the firewire camera. The VideoGrabber has to grab a video image and write it into a shared memory block.
- MarkerDetection
  The MarkerDetection reads the video data from the shared memory block an then calls the ARToolkit functions to detect markers in the image. If a marker has been detected an event must be produced. The event contains the marker-ID and a confidence value which gives the quality of the detection of the marker.

- ContextEstimator
  The ContextEstimator listens to the marker events and carries the current context information. It has to transform these marker events into high-level context information. The configuration of this component is not fixed. The ContextEstimator adapts during the runtime to the current situation. So we have to solve two questions:

  – How can we carry the context information?
  – How can we have a dynamic configuration during the runtime?

  We solved the first question as we are using a simple map. The map has the marker events as keys and the new context situations as values.

ContextEstimator with current context	
Marker-Event m	Context n
Marker-Event l	Context k
...	...

Table 3.2: Context Estimator with context information

We now see that this map only carries the information for one situation. The map has to be filled with different configurations for each situation. Each time the context changes the map has to be configured again. To change a context different strategies can be chosen [9]. The problem is how to trigger a context change and when should the system be configured with new data.

The second question is how to handle this dynamic configuration of the map. Therefore we need another two services that are responsible for the configuration of the ContextEstimator. But not only this service has to be configured with the right context information. The MarkerDetection has to carry the marker information. Each time the context changes it has to be reconfigured with new marker data. These new services have to provide data and configure the system:

– Configuration - Service
  This service provides access to the data model. The data is stored in a database. The Configuration service was part of a diploma thesis by Marcus Tönnis [7].
– MarkerLoader
  This service carries the current context. It configures the MarkerDetection with the right marker data from the Configuration service and is responsible for the reconfiguration of the ContextEstimator. The MarkerLoader has to make sure that the ContextEstimator carrying the current room context is running.

Later in this thesis we will see that we are able to do an abstraction of this component setup to reuse it for other tracking based systems.

3.2 Overview of the whole system

First we will give an overview of the whole system and then go into detail.

3.2.1 A view on the module

Figure 3.1 shows an overview of the location module. You could see the needs (half circles) and abilities (full circles) of each service and their dependencies among each other.

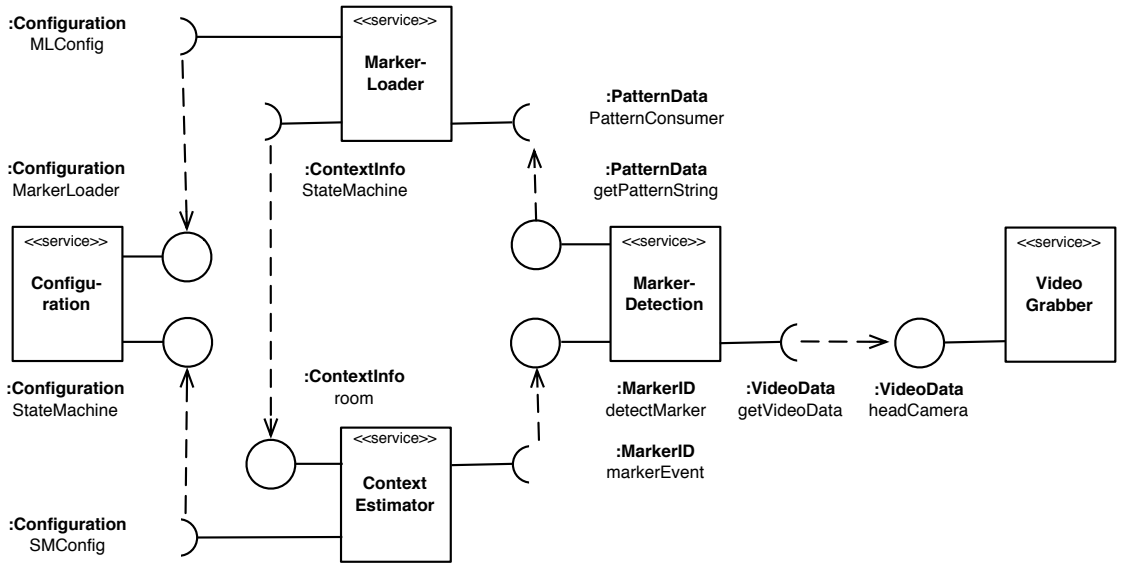

Figure 3.1: Overview of the system

3.2.2 Dependencies between the services

Dependencies are the needs and abilities of the services (see figure 3.3). The following connections are established:

Ability Service	Need Service	type	protocol
VideoGrabber	MarkerDetection	VideoData	Shared Memory
MarkerDetection	ContextEstimator	MarkerID	Events
Configuration	ContextEstimator	Configuration	Method call
ContextEstimator	MarkerLoader	ContextInfo	Method call
Configuration	MarkerLoader	Configuration	Method call
MarkerLoader	MarkerDetection	PatternData	Method call

Table 3.3: VideoGrabber - MarkerDetection

- VideoGrabber - MarkerDetection
  The VideoGrabber has the ability for VideoData. The MarkerDetection has the need for VideoData. According to the basic DWARF mechanisms the servicemanager connects both services with each other. They are exchanging the video data via shared memory. One restriction is that both services have to run on the same machine (there is no global shared memory available).

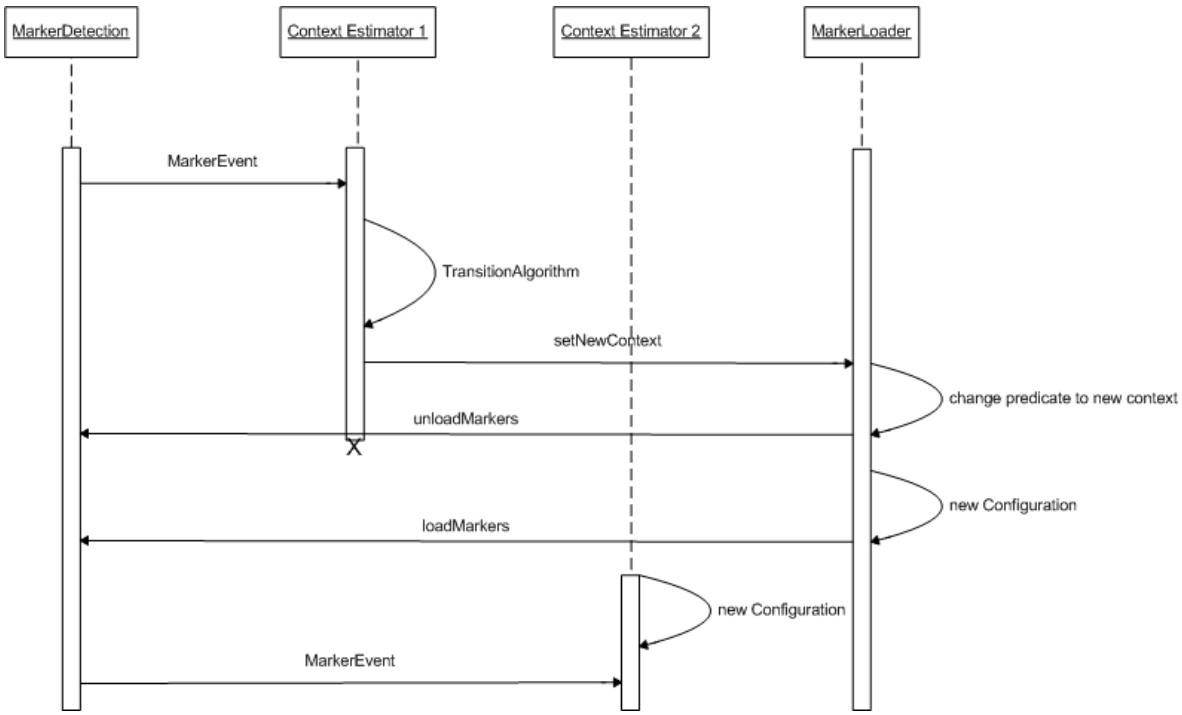

Figure 3.2: Sequence diagram

- MarkerDetection - ContextEstimator
  The MarkerDetection sends a marker event to the ContextEstimator. The event contains the marker ID and a confidence value.
- Configuration - ContextEstimator
  The ContextEstimator gets the proper context configuration from the configuration service.
- ContextEstimator - MarkerLoader
  If a context change occurs the new context is set in the MarkerLoader.
- Configuration - MarkerLoader
  The MarkerLoader gets the marker data from the database.
- MarkerLoader - MarkerDetection
  The MarkerLoader configures the MarkerDetection with a set of marker. These markers are depending on the current room context. In every room other markers are active and needed.

3.3 Interaction of the Services

In figure 3.2 you can see how the services interact with each other. Each time the context changes the module runs through certain steps:

1. The MarkerDetection detects a marker and send the marker event to the ContextEstimator
   The MarkerDetection creates an event containing the marker information about the detected marker. This information contains the current marker ID and a confidence value describing the quality of the detection.

2. Context transition by the ContextEstimator
   The ContextEstimator now has to trigger the transition to a new context caused by the incoming marker event. Therefore a strategy has to be chosen. This strategy is a transition algorithm defining the behavior of the ContextEstimator. For example a marker has to be detected by the MarkerDetection ten times before the ContextEstimator triggers the transition. There are many possible strategies also depending on the current environment. Possible strategies are described later in this thesis.

3. Set new context in MarkerLoader by the ContextEstimator
   The new context is set in the MarkerLoader service. This is done by a simple method call on the MarkerLoader.

4. Change predicate to new Context in MarkerLoader
   As the MarkerLoader is responsible for the configuration of the MarkerDetection and the ContextEstimator it now has to change the configuration of both services. The MarkerLoader changes its need predicate for the configuration and it changes its need for a ContextEstimator. The old ContextEstimator does not fit anymore to the need of the MarkerLoader, it is of no use and it dies. This is described later in the chapter dynamic configuration (3.4).

5. Unload markers in MarkerDetection
   Before the MarkerDetection can be configured with new marker data, the old markers have to be unloaded first.

6. New configuration of the MarkerLoader
   The MarkerLoader first has to change its own configuration in order to provide the right maker data for the MarkerDetection. This means the MarkerLoader has to query the Configuration service (database) for the new data. This is also described in the section dynamic configuration.

7. Load markers in MarkerDetection
   Now the new marker information is loaded into the MarkerDetection. The MarkerDetection is now configured to the new context.

8. New ContextEstimator is started
   The need of the MarkerLoader for a ContextEstimator has changed in step 4. Now a new ContextEstimator is started according to this need. The ContextEstimator has a "startOnDemand" functionality provided by the servicemanager. Now it is "demanded and started".

Now the whole system has changed its configuration for the new context situation. A simple example would be if a user just leaves a room and detects the marker while exiting it. The system initiates a context change and adjusts to the outside of the room. A whole example

is shown in the scenario chapter 4. In the next subsection it is shown how the dynamic configuration is done.

Now it is possible to consider the ContextEstimator as a service that provides a state machine. Every context is one state. So we now say the ContextEstimator is a state service managing the transitions out of the state into the next one. If you put all the state services together we have a state machine consisting of all possible context situations. We will now call the ContextEstimator **StateService**.

3.4 Dynamic configuration: MarkerLoader and StateService

3.4.1 MarkerLoader

As we have already said, the MarkerLoader is carrying the context of the current situation. It has to load the marker data into the MarkerDetection. Therefore it has to query the database (configuration service) for the marker information.

The service description has a need for configuration. The ability for this need is provided by the configuration service:

```
<need name="MLConfig" type="Configuration"
      predicate="(&(configurationKey=MarkerLoader)
      (room=StartRoom))">
       ...
      <connector protocol="ObjrefImporter" />
</need>
```

Figure 3.3: MarkerLoader - Need description for configuration

In section 1.2 we described the possibility of setting predicates. In this case it means that the MarkerLoader will only connect to a Configuration service with the ability "Configuration" that carries the attributes <attribute name="configrationKey" value="MarkerLoader"> and <attribute name="room" value="StartRoom">. This is needed to guarantee that only the information for the "StartRoom" is loaded. As every possible service is able to use the Configuration database you have to set a configuration key.

The database has now the a table that has all the queried information (see figure 3.4).

ConfigurationKey	**Room**	**Key**	**Value**
MarkerLoader	StartRoom	MarkerSet	1/3/4
MarkerLoader	StartRoom	1	<PatternData>
MarkerLoader	StartRoom	3	<PatternData>
MarkerLoader	StartRoom	4	<PatternData>

Table 3.4: Database table for Configuration

While initiating the MarkerLoader it get a reference on the Configuration service. Now it first has to call the "getProperty" - method to receive the marker set of the current room.

This only shows which markers are available to perform a context transition out of the state. After the query for the marker set, the MarkerLoader now queries for the pattern data.

After all the pattern data is queried it now calls the ”loadPattern”-method on the MarkerDetection reference. For each marker the method is called. Of course the loaded marker IDs are stored in order to unload them after a context change occurs. Now the MarkerDetection is configured and enabled to detect marker in the current room.

3.4.2 StateService

In the interaction of the services we have seen that the StateService is started and shut down automatically due to the need description of the MarkerLoader (see figure 3.4).

```
<need name="StateService" type="RoomInfo" predicate="(room=StartRoom)">
 ...
</need>
```

Figure 3.4: MarkerLoader - Need description for a StateService

The StateService offers the corresponding ability for this need. The StateService also has a need for its configuration. This configuration is the information about the current state. The service provides information about the outgoing state transitions. Therefore this data must be queried. In order to start and configure the StateService dynamically the DWARF system provides a *-functionality. In the need and ability description of the StateService you are able to hand over values (see figure 3.5).

```
<ability name="Room" type="RoomInfo">
   <attribute name="room" value="$Config.room" />
    ...
</ability>

<need name="Config" type="Configuration"
    predicate="(configurationKey=StateService)">
    <attribute name="room" value="*" />
      ...
      <connector protocol="ObjrefImporter" />
</need>
```

Figure 3.5: StateService - The star attribute [6]

The need ”Config” carries an attribute <attribute name=”room” value=”*” />. This means that every possible configuration (for every room) is offered to the StateService (see figure 3.6). As the Configuration service is running all the time all the configuration can be selected right now. All of these values are now put as value in the attribute ”Room” of the ability ”Config”. In simple words all possible StateServices are now offered to the MarkerLoader that carries the need for a StateService. In this example the MarkerLoader has the need with a predicate (room=StartRoom). As the StateService offers all possibilities it also provides the ability with the attribute (room=StartRoom). Now the StateService with this ability is started (see figure 3.7).

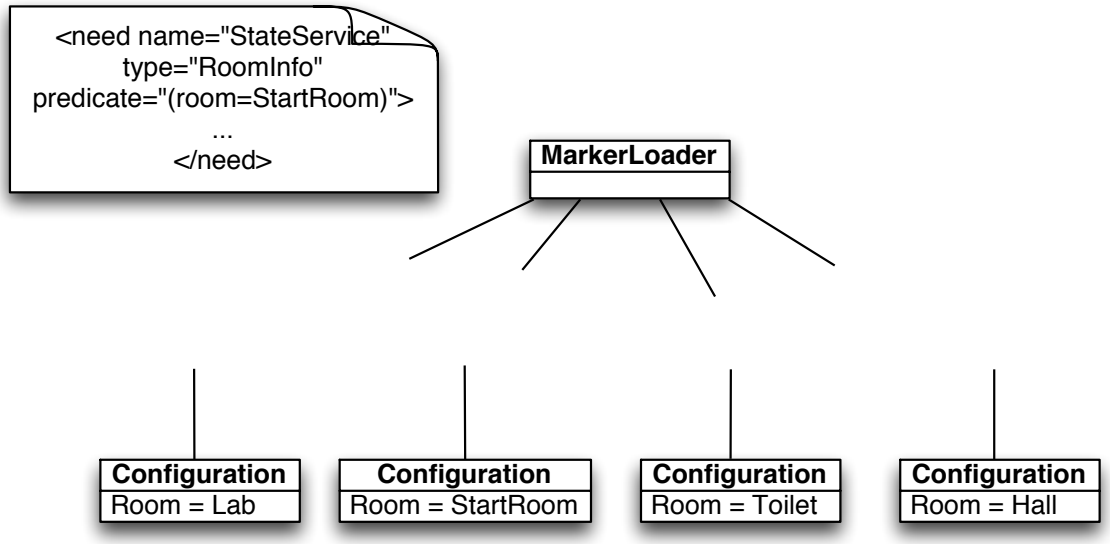

Figure 3.6: Dynamic Configuration Step 1

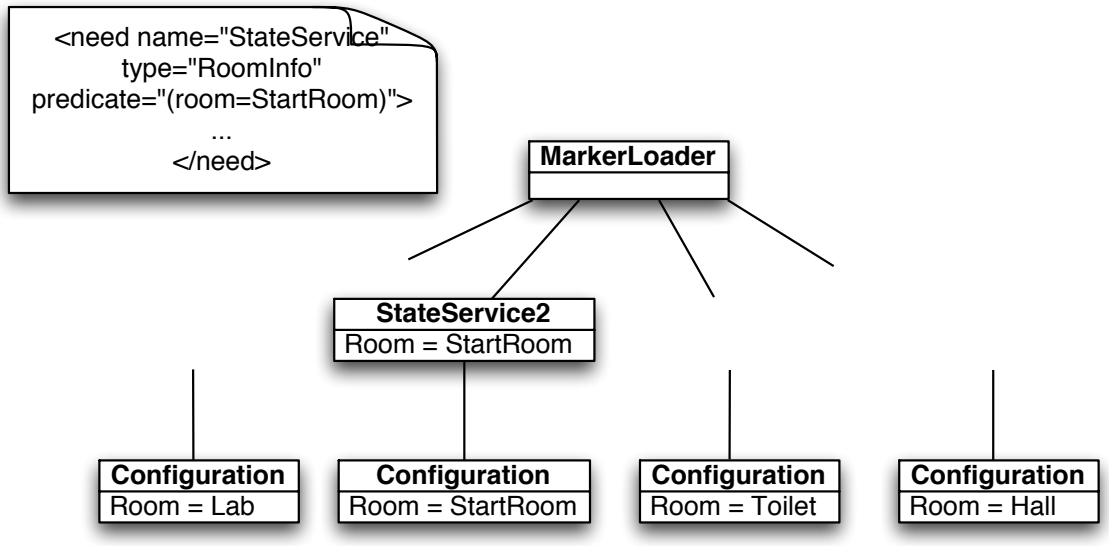

Figure 3.7: Dynamic Configuration Step 2

Due to a context change in the MarkerLoader (room=Hall) the StateService does not fit anymore to the context situation (see figure 3.8). Now the old StateService dies and a new StateService is started (see figure 3.9).

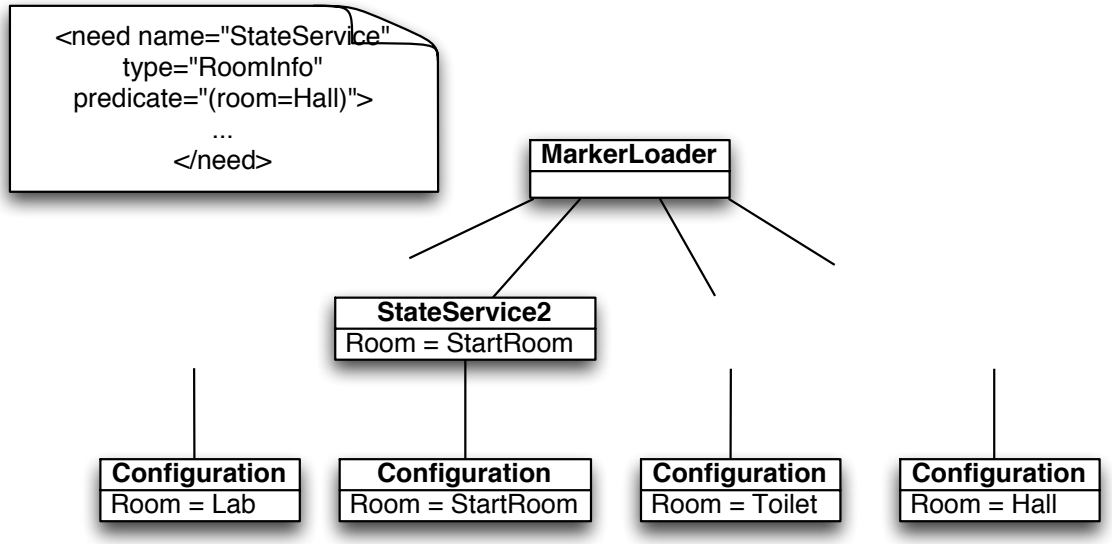

Figure 3.8: Dynamic Configuration Step 3

Now the system is configured to the new context. The advantage of this * - attribute functionality is that the StateService is able to configure itself to the context. Otherwise it would be necessary to design several StateServices for every state manually. The service description of this StateService allows to start the StateService with different configurations.

The configuration of the StateService itself works the same way the MarkerLoader does the configuration. It first queries the existing markers in the current room. The second step is to read the context transitions from the markers to new rooms (according to figure 3.2).

3.5 Tracking: VideoGrabber and MarkerDetection

In this section the design and functionality of the services that are responsible for tracking is explained.

3.5.1 VideoGrabbber

The VideoGrabber is based on the dc1394 library. We will not go into detail of this library. But we will explain the detailed architecture of the VideoGrabber.

1394-Based Digital Camera Control Library

As mentioned the VideoGrabber is based on the libdc1394 library. Here is a small overview of the functions the VideoGrabber uses. For further information about this library have a

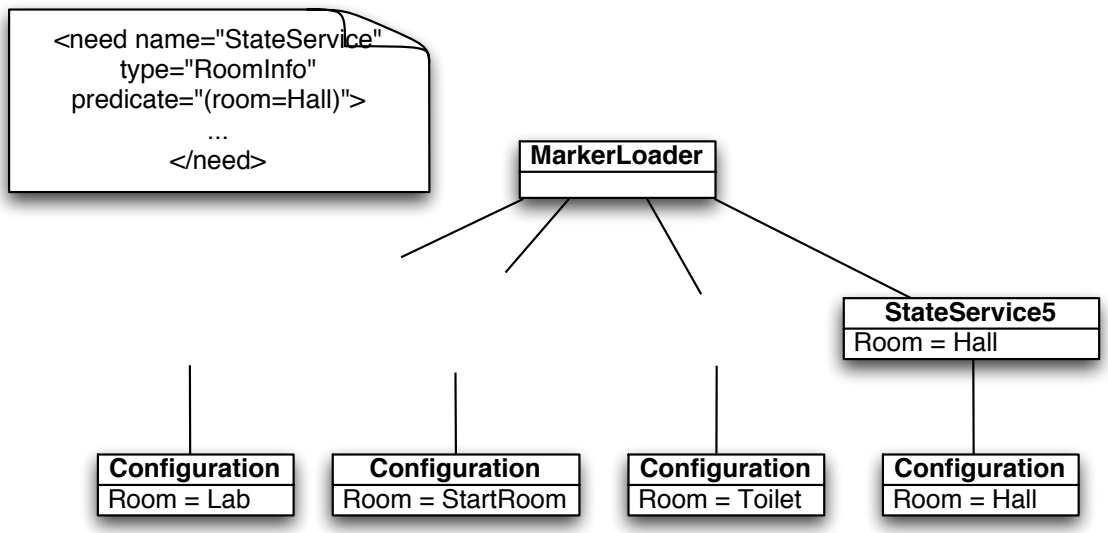

Figure 3.9: Dynamic Configuration Step 4

look at the dc1394 control.h header file.

The libdc1394 is a library for the use of digital cameras via a linux firewire connection. It delivers several methods to handle the use of a camera. Also a direct memory access via a DMA channel is supported. The VideoGrabber component uses this kind of data transfer because it is much faster. The library is built on the libraw1394 to manage the firewire connection.

Architecture of the Video Grabber

The following UML-diagrams are describing the static structure, the communication between the objects and the internal life cycle of a VideoGrabber-object.

1. **Static structure**

   The VideoGrabberService controls the VideoGrabber (see figure 3.10). As the VideoGrabberService is a DWARF-Service it reads abilities for every camera. This ability has the type "CameraData". It instantiates a VideoGrabber and hands over the ability description to the VideoGrabber. For every new "CameraData" ability the VideoGrabberService starts a new thread for the life cycle of the VideoGrabber. So every VideoGrabber runs in parallel. If the VideoGrabber stops it will be deleted by the VideoGrabberService as well.

   In the service description (see figure 3.11) you can see that it is possible to hand over certain attributes describing the abilities of the camera. If these attributes are set to zero a standard configuration is loaded. These values are integer values describing each value. These values are specified in the dc1394 control.h. The standard configuration uses a

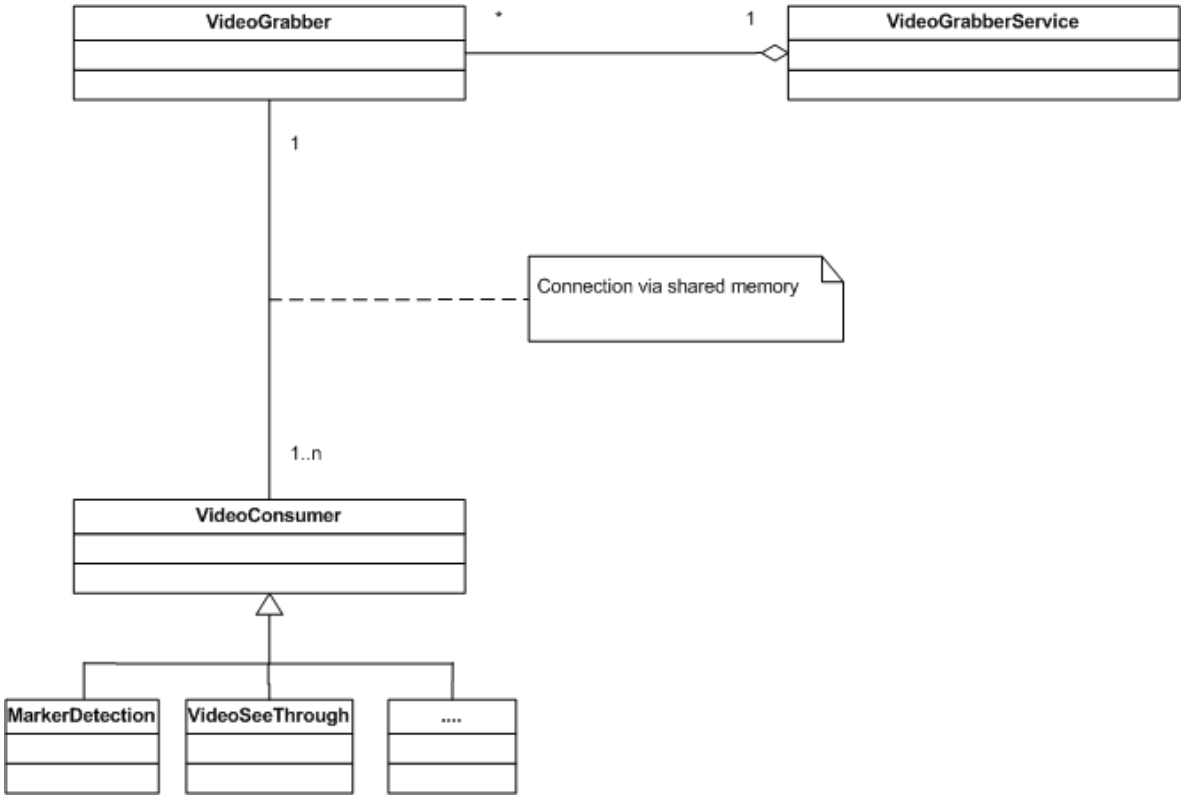

Figure 3.10: VideoGrabber UML Class-Diagram

```
<service name="VideoGrabberService">
       <ability name="sendHeadCameraData" type="CameraData">
            <attribute name="node" value="0"/>
            <attribute name="mode" value="0"/>
            <attribute name="rate" value="0"/>
            <attribute name="channel" value="0"/>
            <attribute name="speed" value="0"/>
            <attribute name="format" value="0"/>
            <attribute name="dmabufnum" value="0"/>

            <connector protocol="Shmem"/>
      </ability>
</service>
```

Figure 3.11: VideoGrabber - service description

640x480 video mode with YUV411 colors and a frame rate of 30. These parameters are optimal when using an iBot-camera. If another camera is added the service description would get another ability carrying the proper attributes.

The VideoGrabber component offers images via shared memory. Every other service having the need for "CameraData" (in our case: the MarkerDetection) could access the shared memory and use the image for different purposes. These are the consumer services.

2. **Sequence Diagram**

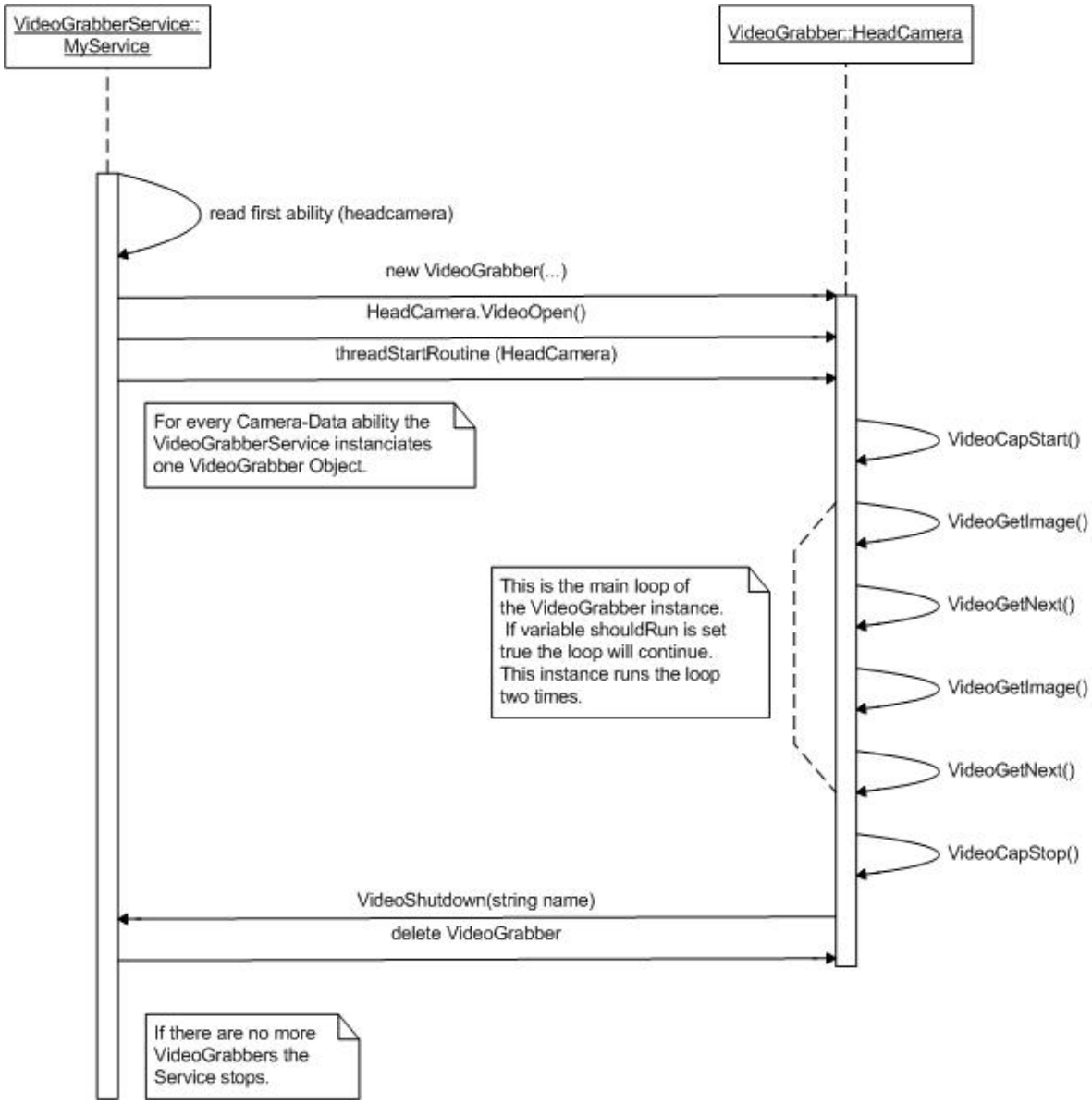

Figure 3.12: VideoGrabber UML Sequence Diagram

If a VideoGrabberService is started the startService method is called by the service-manager. It reads the abilities specified by the external xml service description. As mentioned before each connected camera is described in this file. If the service reads a new ability description it starts a new thread and calls the controlGrabber method. This method controls the life cycle of each VideoGrabber object (see figure 3.12).

a) Instantiation of the VideoGrabber Object
A new VideoGrabber object is instantiated. The parameters for the video capture are given to the constructor. These parameters are set in the ability description of the camera.

b) Establish the connection to the camera
A connection to the firewire card is opened and the camera gets initialized.

c) Start the transmission
A DMA (Direct Memory Access) channel is opened and the camera is ready for transmission. This channel makes it possible to bypass the processor. This is much faster that the usual method.

d) Grab a picture and release it
A video image is grabbed and now written into a buffer. After the image has proceeded the buffer gets released.

e) Stop the transmission
The transmission is stopped if the user stops the service or an exception occurs (e.g. unplugging the camera).

f) Delete VideoGrabber object
If the VideoGrabber object finishes its transmission it gets deleted from the control object (VideoGrabberService).

If a VideoGrabber object runs through all these steps or stops because of an exception it quits its main loop and gets deleted by the VideoGrabberService.

3. **Statechart diagram**

The first method called during the life cycle is the **VideoOpen** method (see figure 3.13). It opens a DMA channel to bypass the processor. This method also queries the camera attributes. The **VideoCapStart** method starts the iso-transmission from the camera. Then the object enters the main loop consisting of the steps **VideoGetImage** and **VideoGetNext**. The first step in the loop gets the image and puts it in a special buffer. Then the buffer is released by the VideoGetNext method. This loop continues as long as the shouldRun variable is set true. It will be set false if an exception occurs or the controlling VideoGrabberService calls the stopRun method. After the main loop is finished the **VideoCapStop** method stops the iso-transmission and releases the camera.

3.5.2 MarkerDetection

This component is based on the ARToolkit. It gets images from the shared memory block and searches for marker information in it. It then creates an event carrying the marker information of the detected marker.

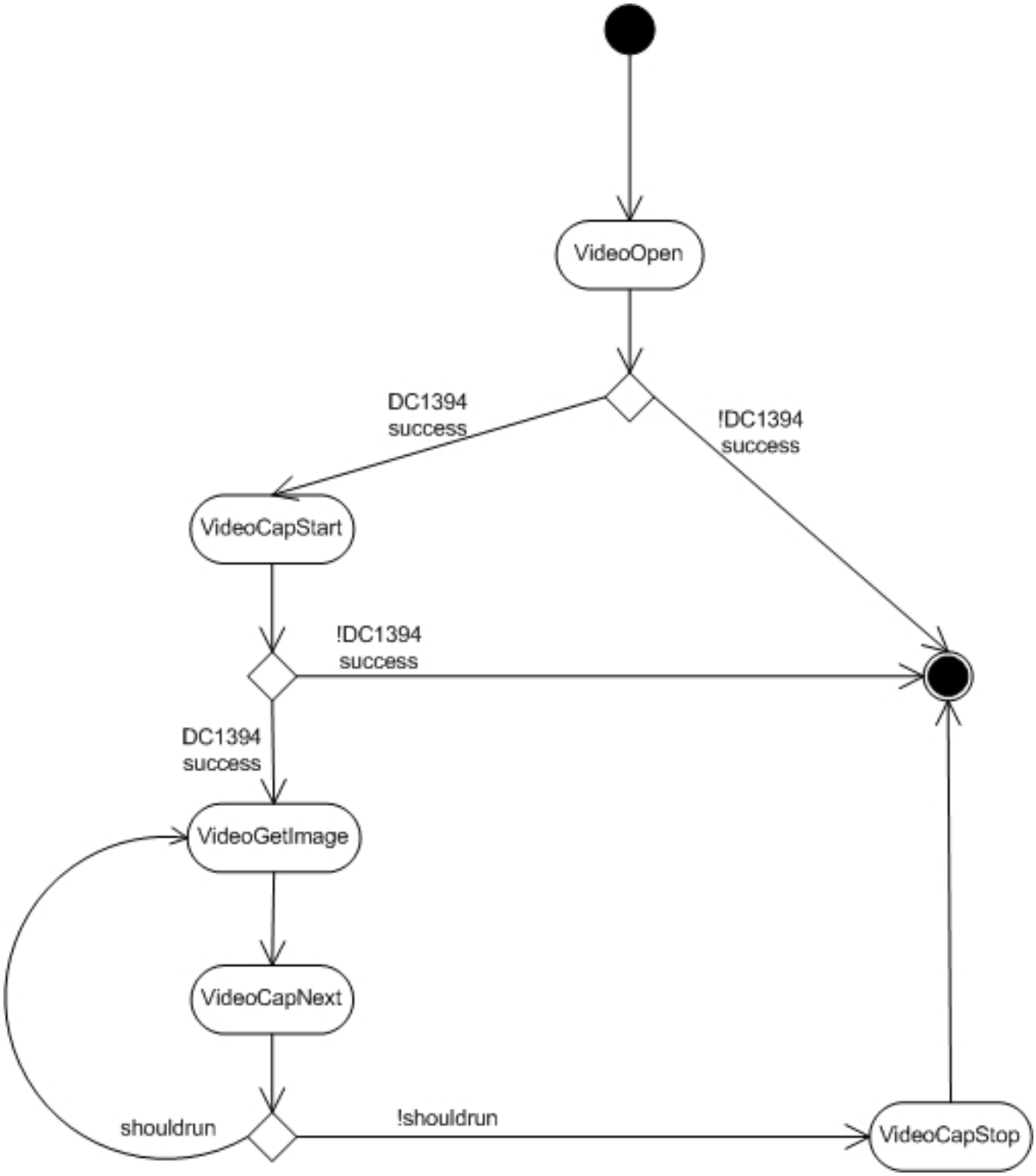

Figure 3.13: VideoGrabber UML Activity diagram

```
<service name="MarkerDetection" startCommand="MarkerDetection" isTemplate="true">
        <need name="readHeadCameraData" type="CameraData">
              <connector protocol="Shmem"/>
        </need>

        <ability name="acceptPatternString" type="PatternData" >
               <connector protocol="ObjrefExporter" />
        </ability>

        <ability name="detectMarker" type="MarkerID">
                <connector protocol="PushSupplier"/>
        </ability>
</service>
```

Figure 3.14: VideoGrabber - service description

In the service description (see figure 3.15) you could see that the service has the need for "CameraData" for the connection with the VideoGrabber, the ability to accept patterns for the dynamic configuration and the ability to send marker events to a consumer. Accepting patterns means that patterns can be loaded in the MarkerDetection via a method call. This job is done by the MarkerLoader service as we have already explained.

The following steps are done during a context period. Context period means that the MarkerDetection has to be configured according to the current context. Configuration data is the number of patterns and the pattern data. If the context carries 3 markers, then the MarkerDetection has to be configured with the pattern data belonging to these 3 markers. After a context change these makers have to be unloaded first, then the component can be configured with the new context information again.

1. Loading the pattern data
   The MarkerDetection provides the method loadPattern:

   ```
   loadPattern(const char *pattern, const char *id, bool finished)
   ```

   The first parameter is the pattern data. Each marker has a unique ID, this is the second parameter. The last boolean parameter is only an information if the last marker is loaded. The pattern data is stored in a temporary file and then initialized with the ARToolkit method:

   ```
   int arLoadPatt((char*)tmpfile)
   ```

   The integer return value is the internal marker ID of the ARToolkit. So we have to keep a map using the unique ID as key and the ARToolkit ID as value (see 3.5). After the last marker is loaded (boolean finished) the MarkerDetection is configured and ready.

Marker ID	**ART ID**
1	0
5	1
3	2

Table 3.5: Example: MarkerID Map

2. Getting the image
   The MarkerDetection reads the shared memory block.

3. Detecting markers
   After the image is read out of shared memory the following ARToolkit method is called:

   `int arDetectMarker((ARUint8*)image, int thresh, ARMarkerInfo &markerinfo, int &markernum )`

   The first parameter is the image (it is casted into unsigned char format), second is the threshold for the detection. The method writes its result into the struct markerinfo. This is a struct (type ARMarkerInfo) carrying an array with the detected markers and their information. It is possible that more than one marker is detected. If two markers are detected, markerinfo carries two marker informations and markernum is set to 2.

   Now we can run through the markerinfo array and compare if one of our loaded markers fits to detected one. This is realized in two loops: The outer loop is running through the array of detected markers (see figure 3.15).

```
for( j = 0; j < marker_num; j++ )
   {
       // The inner loop compares the ids of the markers:
       // m_hashmap is the markerID map

       for(pos = m_hashmap.begin(); pos != m_hashmap.end(); ++pos)
          {
              int patt_id = pos->second;

            // patt_id is the internal ARToolkit ID
         ...

         if( patt_id == marker_info[j].id)
           {
           // a marker has been detected
           }
           ...
         }
    }
```

Figure 3.15: MarkerDection: detectMarker-loop

   If more than one marker is detected the most likely marker is chosen. The markerinfo also carries a confidence value. The value is compared and the marker with the higher one is taken. If it is necessary to detect two markers at the same time this method has to be adapted for that purpose.

4. Sending the event
   If a match occurs an event is produced. The event carries the unique marker ID. Now the event consumer is able to get the value of the detected marker ID (in our case this is the StateService).

4 Environment setup

This chapter shows an example scenario setup and how the system gets configured according to this example. It will also show a new problem how to deal with context transitions. We will also have a look at the hardware setup of the mobile client.

4.1 Test environment

In figure 4.1 you can see how the rooms are ordered in our test environment. The user moves through this rooms and on every door to a new room a marker is attached nearby.

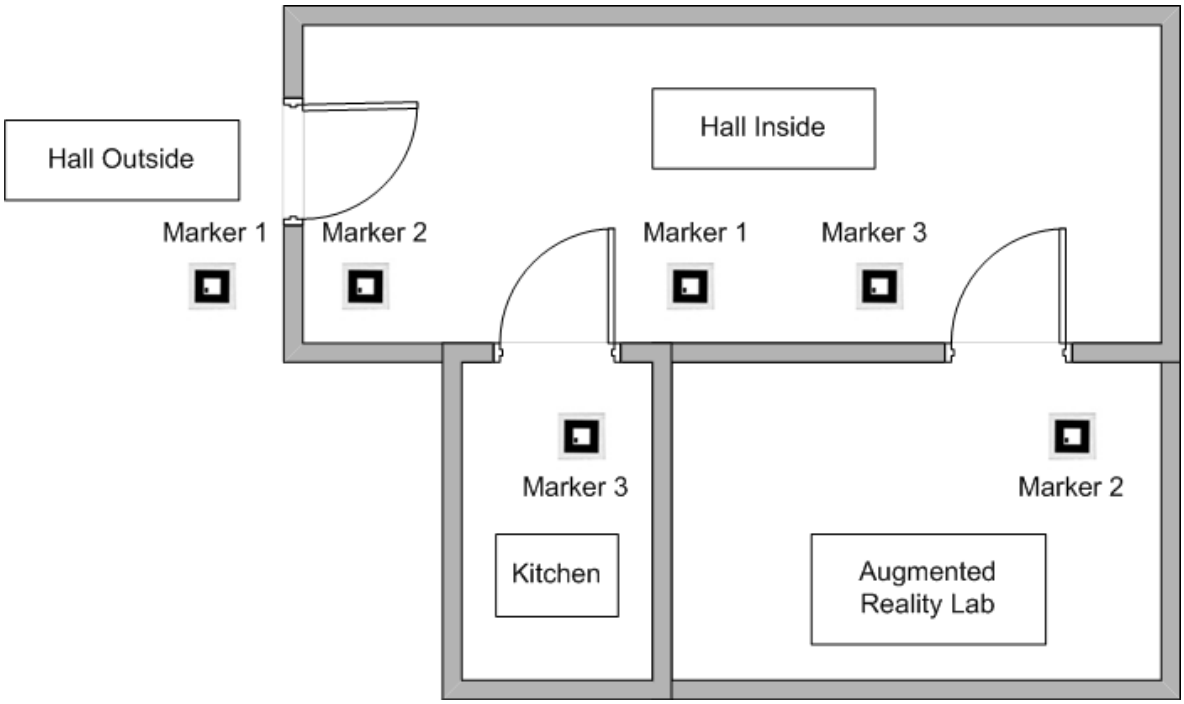

Figure 4.1: Environment

The whole setup of the rooms could be modeled a one state machine now (see figure 4.2). Every transition is labeled with the the ID of the transition marker. For example if the user moves from the hall inside to the kitchen the marker with ID 1 triggers the transition to the new state. As we have already said the StateService only carries the information of one state. For each state of this state machine a StateService with the connected context is available. Note that it is possible to reuse markers. In this example every marker is used twice.

The StateMachine has a simple hash map with the marker ID as key and the new room as value. For example the hash map for the state "Hall Inside" shows all possible transitions

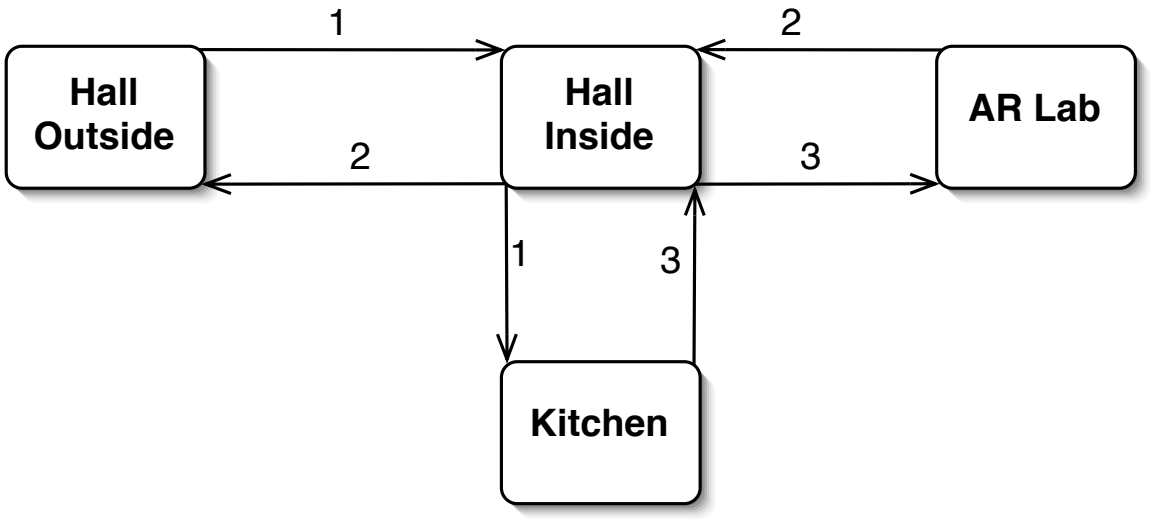

Figure 4.2: State machine

(see figure 4.1).

transition marker ID	new state
1	kitchen
2	Hall Outside
3	ARLab

Table 4.1: StateService hash map for the state "Hall Inside"

Now if the MarkerDetection sends events caused by a detection of marker with ID 2. The StateService runs through this hash map and triggers a transaction to the new state "Hall Outside". According to the steps described above (see **??**) the system configures itself to the new context. A new StateService now carries the hash map for the state "Hall Outside".

It is easy now to extend the environment with new rooms without editing the old configuration. It is only necessary to put the new transitions from the old environment into the new rooms in the old configuration.

4.2 Triggering the transitions

One thing we have not talked about yet is the task of triggering a context transition. We will now explain the problem of that task and possible strategies in order to have a robust system.

4.2.1 Current Realization

It is a hard problem how the context transition from one state in a new state should be done. The quality of the marker detection is the main problem. The quality depends on a lot of factors that should be taken in account (taken from [9]).

Light conditions: Depending on the light conditions markers will be detected easier or not. Usually in full daylight the marker detection is able to detect markers quite robustly. But the possibility of failure rises. If conditions get darker, the AR Toolkit code tends to classify markers correctly only occasionally.

Movement of the user: The marker detection is also influenced by the users movements. If a user moves quite fast a marker might not be detected due to image blur.

Marker position: If markers are attached to locations which are not in the view of the user markers might also not be detected.

Camera: Depending on the camera properties and the camera calibration the quality of the marker detection varies.

Room properties: If a room is a long hall you could assume the direction of the user and the sequence of markers that will be detected. On the other hand a large room with many possibilities to walk to is more difficult to handle.

The current realization does not consider any of these factors. We have decided to chose a default strategy to detect a marker several times before a state transition is triggered. Our experience shows that 10 marker events deliver an adequate result. But due to different light conditions in each day time the quality of the detection varies. In the evening for example it is quite hard to detect a marker 10 times. On the other and in the morning markers are detected quite easily. But if light conditions are good, wrong markers are detected more often.

4.3 Hardware Setup

The hardware setup consists (see figure 4.3) of the following main components:

The DWARF "Kraxn": All of the hardware components are attached to a rucksack.

iBot camera: The camera is attached to the rucksack near the head of the user. All the markers should be stuck on the wall in the same height in order to get a good quality of the marker detection.

Laptop: The laptop on the back of the rucksack has a wireless LAN connection to the available network node. The tracking component (VideoGrabber and MarkerDetection) as well as the StateService and the MarkerLoader is running on this computer. The Configuration service is running on a machine in the network with the database running on it. On a 1.8 GHz machine with a working load of 80% it took about 5 second to switch between two contexts.

Touch glove: The touch glove is a result of a SEP by Johannes Woehler [11]. It is used in ARCHIE to have a user interface that could be personalized by different user profiles.

Figure 4.3: ARCHIE hardware setup

To make this hardware setup more easy to install a wireless PAN realized by a Bluetooth connection would be an improvement.

5 Conclusions

Finally we want to give a summary and some ideas for improvements of the system.

5.1 Summary

We described a new application domain for the ARToolkit: wide area tracking. We use the optical marker detection functionality not for the purpose of displaying virtual models but for the creation of marker events. With this marker events it is possible to estimate the current context. The key concept of this work is that the whole system gets configured by the environment. As a consequence a mobile user does not need any spacial information on his setup. The mobile client consists of a small computer and one or even more sensoring devices. It carries only the information that is needed at the moment. Due to this we get rid of any update problems of the client. According to the current context the configuration component and the context estimator are appearing in several instances. An example in our system would be that the ContextEstimator would be reconfigured by altering the hash map that provides the outgoing transitions. As we have seen the ContextEstimator dies and a new instance with a new hash map starts up. But also the MarkerLoader which is represented by the configuration component in our architecture changes its configuration caused by a context change. Therefore a mechanism is needed to run components with different setups. In our case this is provided by the DWARF framework. To realize this we described an idea for a system that gets reconfigured all the time:

- Configuration
  A component is needed to get data from the data model but also to provide the necessary context data to either a context estimation or a sensoring component. This component carries the context. We described a dynamic push mechanism in order to reconfigure its components after a context change.

- Sensoring
  The sensoring component is necessary for the gathering of context information. A video image is grabbed and an ARToolkit based detection component searches for marker information in it. This component has to be reconfigured with the right marker information after a context change.

- Context Estimation

  The estimation is able to calculate the approximate context out of the raw tracking data. This calculation can differ from context to context. Therefore it needs also a configuration of its properties. As we have seen it is really a problem how to trigger a context change.

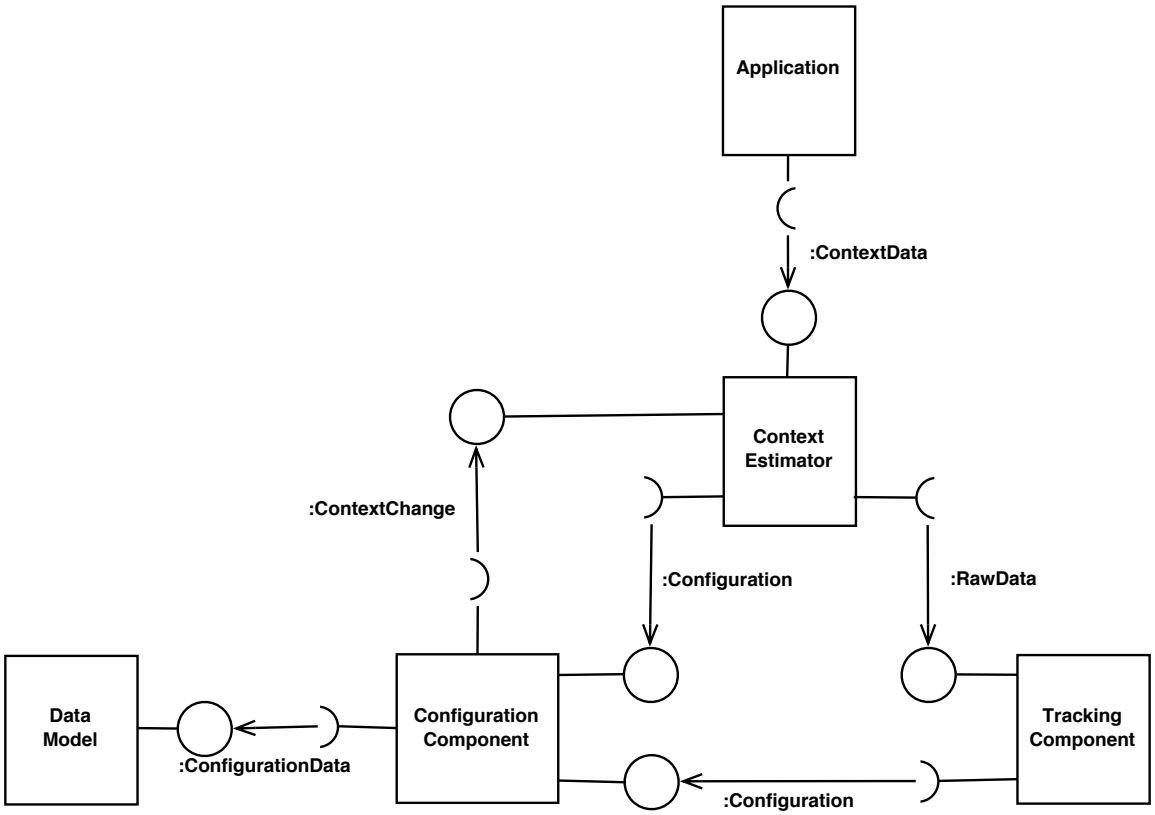

Figure 5.1: Pattern for context aware systems

In figure 5.1 you are able to see the whole layered system with all the need and ability dependencies.

The Context Toolkit proposed by Anind Dey [2] is also an attempt to provide an architecture for context-aware applications. As you can see it looks quite similar (see figure 5.2) to the architecture of our location module.

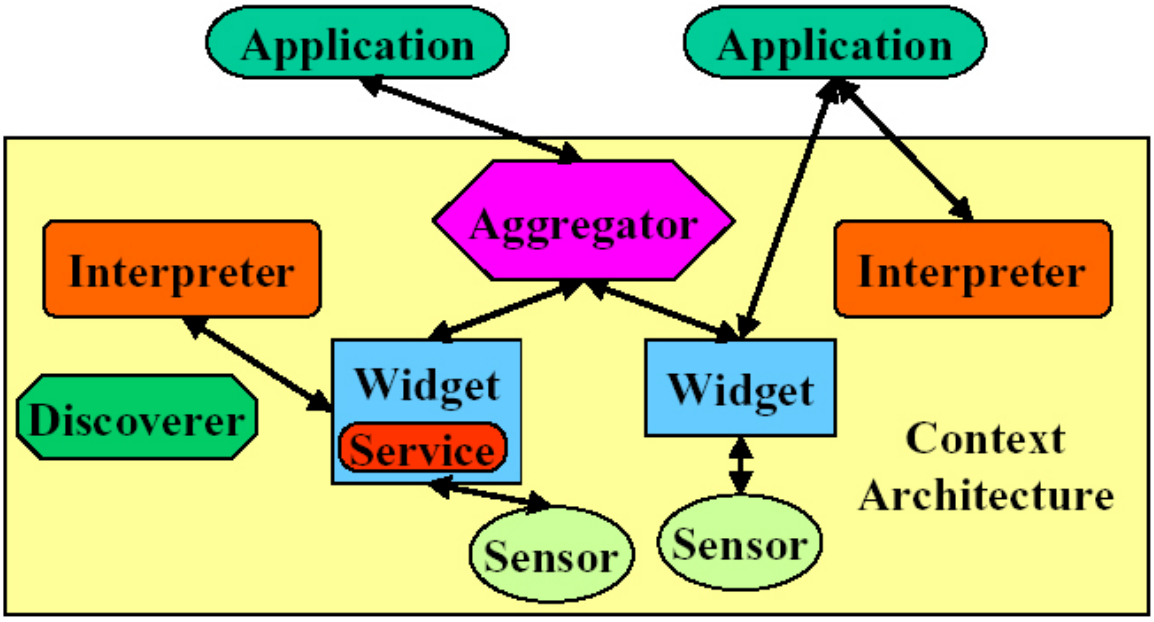

Figure 5.2: Architecture of the Context Toolkit

Sensors give sensoring data to widgets. These widgets can be subscribed by interpreters or by different applications. An interpreter would be our ContextEstimator. This component view does not consider a dynamic configuration. But some key similarities can be seen. For a more detailed view please have a look at the Dey thesis [2].

At least this is a step to create intelligent environments without asking for explicit user input. All the reconfiguration mechanisms are invisible to the mobile user. The only interaction is the selection of an application that is provided to the user. But we also see if we walk toward Marc Weisers vision [10] the systems get more complex and distributed. But these systems must also be more usable.

5.2 Future work

The goal of this thesis was to do a prove of concept, not to build a robust application. Starting from this design, much more work could be done in the future to enhance the robustness of our approach. Due to wrongly triggered transitions caused by wrong marker events the system often gets inconsistent and the ContextEstimator gets configured with wrong data. So we need transition strategies that adapt very much to the current situation a mobile client is in.

The attempt is to gather information from the environment and derive a proper strategy for every room. This strategy might vary for each room (see figure 5.3).

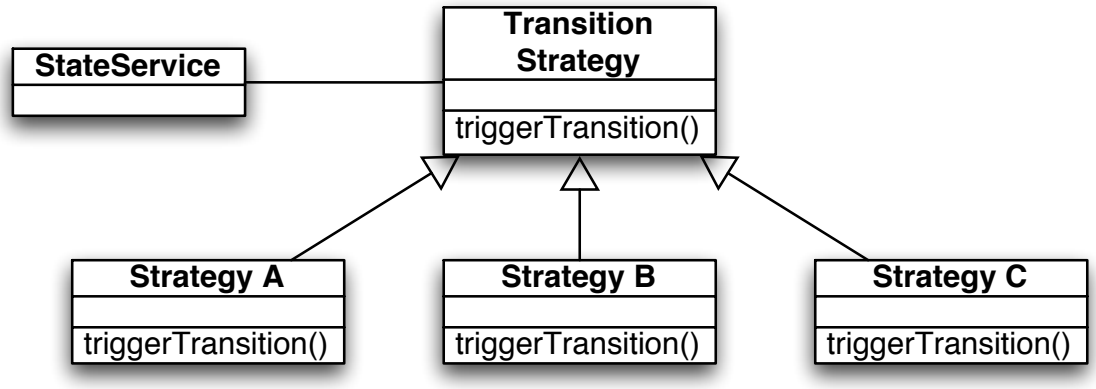

Figure 5.3: Strategy pattern for transition algorithm

Here are some possible strategies (also taken from [9]):

Minimum number of frames with marker: A very basic but often sufficient strategy is to require that a marker is detected several times within a given time frame or number of subsequent image frames. The larger the number of frames gets, the more robust this strategy is. The disadvantage of this strategy is that we have to ensure that the user's camera has the marker to be detected in its field of view for a probably long time. In consequence, the user is restricted in his movements if context changes should be detected correctly. We chose this strategy as a default strategy that was easy to implement.

Weighting number of frames by confidence value: It is also possible to take the confidence value of a marker detection into account. A very simple strategy is to only take markers with high confidence value into account. A more sophisticated strategy is to combine the number of frames and the confidence value. A context change will be triggered if the sum of all confidence values during the last time frame is beyond a given threshold.

Detect several markers at the same time: Another strategy is to attach two markers next to each other, initiating a context change only together. A transition is only triggered if both markers are detected simultaneously within a given time frame. Obviously, this method can be extended to an arbitrary large number of markers.

Sequence of markers: Finally, we can define a sequence of markers triggering the transition. In the most simple setup, a 'leave room' marker is attached in the current room and a 'confirm' marker is attached in the new room. Only if the *MarkerDetection* detects the confirm marker after the leave room marker, a change of context is triggered. Again, this strategy can be extended to a larger sequence, which may be well suited if the user must follow a fixed path to get from one room to another.

All of these strategies can be combined with each other to yield even more robust strategies. In addition, all strategies have parameters (such as the minimum number of frames) that can be fine tuned. These parameters may even be adjusted automatically depending on other contextual information. For example, the current lighting conditions may influence the choice of confidence value, as the AR Toolkit code usually yields better results in brighter environments. If we have no means to estimate the current lighting conditions automatically, we may use the current time of day to get a rough estimate as shown in table 5.1.

Light conditions	**Number of frames**
good: full daylight	high (e.g 20-30)
medium: normal daylight	middle (10-20)
bad: no daylight, halogen lamps	low (5-10)

Table 5.1: Light conditions and number of frames

Another attempt would be to take some machine learning algorithms into account. It should be possible to train and tune your system in order to receive a high quality marker detection. Probably machine learning algorithms that are able to detect and revert a wrong transition are needed. Another idea would be to allow the user to revert a transition. Of course this is not what we want to, but in order to avoid inconsistency and to build a robust system it would be a possibility. This would be leading to a semiautomatic configuration of the context estimation component. In order to validate our architecture it should be used for other applications that support a dynamic configuration for mobile clients. Other tracking mechanisms can be used. Interesting would be the context estimation out of multiple tracking resources. This would also be a contribution to solve the problem of wrong transitions. The more context we are able to consider to estimate our high level context the better would be the estimation.

But also improvements concerning the single components could be done:

- VideoGrabber
  The VideoGrabber now only supports one camera attached to the fire wire port. To improve the marker detection two or more cameras would be good. One strategy could be that two cameras have to detect a marker several times to perform a context transition. The design of the VideoGrabber allows multiple cameras (e.g. a camera attached to the head, another attached to the breast). But this has to be tested and enhanced. The attributes of the video grabbing should be free configurable (frame rate, format, ...)
- Configuration (data model)
  This service supports a very simple interface for DWARF services. According to the ConfigurationKey all key-value informations are stored and provided to each service. The problem is that the Configuration service does not support shared resources yet. If two services need the same information, the data is stored twice. This information is redundant.

But the most challenging task is to integrate features and algorithms to make the system much more robust.

Acknowledgments

The author would like to thank Martin Wagner, who advised this work. Special thanks go to all people who participated in the ARCHIE project. It was great fun ;-)

Bibliography

[1] M. Bauer, B. Bruegge, G. Klinker, A. MacWilliams, T. Reicher, C. Sandor, and M. Wagner. Design of a Component-Based Augmented Reality Framework. In *Proceedings of the International Symposium on Augmented Reality*, New York, NY, 2001.

[2] A. K. Dey. *Providing Architectural Support for Building Context-Aware Applications*. PhD thesis, Georgia Institute of Technology, 2000.

[3] DWARF Homepage. http://www.augmentedreality.de.

[4] J. Johannes Wöhler, C. Kulas, M. Kurzak, M. Tönnis, B. Zaun, O. Hilliges, F. Löw, and F. Strasser. Archie: Augmented Reality Collaborative Home Improvement Environment. Projekt, Technische Universität München, 2003.

[5] H. Kato, M. Billinghurst, and I. Poupyrev. *ARToolKit version 2.33 Manual*, 2000. Available for download at http://www.hitl.washington.edu/research/shared_space/download/.

[6] A. MacWilliams and R. Thomas. Decentralized Coordination of Distributed Interdependent Services. Technische Universität München, 2003.

[7] M. Tönnis. Data Management for AR Applications. Master's thesis, Technische Universität München, 2003.

[8] M. Wagner. Building Wide-Area Applications with the AR Toolkit. In *The First IEEE International Augmented Reality Toolkit Workshop*, 2002.

[9] M. Wagner and F. Loew. Configuration Strategies of an AR Toolkit-based Wide Area Tracker. In *The First IEEE International Augmented Reality Toolkit Workshop*, 2003.

[10] M. Weiser. The Computer for the 21st Century. Scientific American. *Scientific American*, pages 94–104, September 1991.

[11] J. Wöhler. Driver Development for TouchGlove Input Device for DWARF based Applications. Systemenwicklungsprojekt, Technische Universität München, 2003.